Programming in C and C++
5. C++: Overloading, Namespaces, and Classes

Dr. Neel Krishnaswami
University of Cambridge
(based on notes from and with thanks to Anil Madhavapeddy, Alan Mycroft, Alastair Beresford and Andrew Moore)

Michaelmas Term 2016-2017
C++

To quote Bjarne Stroustrup:

“C++ is a general-purpose programming language with a bias towards systems programming that:

▶ is a better C
▶ supports data abstraction
▶ supports object-oriented programming
▶ supports generic programming.”

Alternatively: C++ is “an (almost upwards-compatible) extension of C with support for: classes and objects (including multiple inheritance), operator overloading, exceptions and templates”. [templates are a generalised form of generics] Much is familiar from Java, but with many subtle differences.
C++ fundamental types

- C++ has all the fundamental types C has
  - character literals (e.g. ‘a’) are now of type char
- In addition, C++ defines a new fundamental type, bool
- A bool has two values: true and false
- When cast to an integer, true → 1 and false → 0
- When casting from an integer, non-zero values become true and false otherwise
- This is also available in C (since C99) via the <stdbool.h> header
C++ enumeration

Unlike C, C++ enumerations define a new type; for example:

```cpp
enum flag {is_keyword=1, is_static=2, is_extern=4, ... }
```

When allocating storage for an enumeration, you use its name; for example:

```cpp
flag f = is_keyword
```

Implicit type conversion is not allowed:

```cpp
f = 5; //wrong f = flag(5); //right
```

The maximum valid value of an enumeration is the enumeration’s largest value rounded up to the nearest larger binary power minus one.

The minimum valid value of an enumeration with no negative values is zero.

The minimum valid value of an enumeration with negative values is the nearest least negative binary power.
C++ supports references, which provide an alternative name for a variable.

Generally used for specifying parameters to functions and return values as well as overloaded operators (more later).

A reference is declared with the & operator; for example:

```cpp
int i[] = {1,2}; int &refi = i[0];
```

A reference must be initialised when it is defined.

A connection between a reference and what it refers to cannot be changed after initialisation; for example:

```cpp
refi++; //increments value referenced
```
References in function arguments

- When used as a function parameter, a referenced value is not copied; for example:
  ```c
  void inc(int& i) { i++;} //bad style?
  ```
- Declare a reference as `const` when no modification takes place
- It can be noticeably more efficient to pass a large struct by reference
- Implicit type conversion into a temporary takes place for a `const` reference but results in an error otherwise; for example:
  ```c
  float fun1(float&);
  float fun2(const float&);
  void test() {
    double v=3.141592654;
    fun1(v); //Wrong
    fun2(v);
  }
  ```
- Cf. Fortran call-by-reference
Overloaded functions

▶ Functions doing different things should have different names
▶ It is possible (and sometimes sensible!) to define two functions with the same name
▶ Functions sharing a name must differ in argument types
▶ Type conversion is used to find the “best” match
▶ A best match may not always be possible:

```c
1  void f(double);
2  void f(long);
3  void test() {
4    f(1L);  //f(long)
5    f(1.0); //f(double)
6    f(1);   //Wrong: f(long(1)) or f(double(1)) ?
```
Scoping and overloading

- Functions in different scopes are not overloaded; for example:

```c
  void f(int);

  void example() {
    void f(double);
    f(1);  // calls f(double);
  }
```

- Rarely happens in practise since crossing scopes like this leads to confusing control flow.
Default function arguments

- A function can have default arguments; for example:
  ```cpp
double log(double v, double base=10.0);
```

- A non-default argument cannot come after a default; for example:
  ```cpp
double log(double base=10.0, double v);  //wrong
```

- A declaration does not need to name the variable; for example:
  ```cpp
double log(double v, double=10.0);
```

- Be careful of the lexical interaction between `*` and `=`; for example:
  ```cpp
void f(char* =0);  //Wrong ' *= ' is assignment
```
Namespaces

Related data can be grouped together in a namespace:

```cpp
namespace Stack { //header file
void push(char);
char pop();
}

namespace Stack { //implementation
const int max_size = 100;
char s[max_size];
int top = 0;

void push(char c) { ... }
char pop() { ... }
}

void f() { //usage
... 
Stack::push('c');
... 
}
```
Using namespaces

- A namespace is a scope and expresses logical program structure
- It provides a way of collecting together related pieces of code
- A namespace without a name limits the scope of variables, functions and classes within it to the local execution unit
- The same namespace can be declared in several source files
- The global function `main()` cannot be inside a namespace
- The use of a variable or function name from a different namespace must be qualified with the appropriate namespace(s)
  - The keyword `using` allows this qualification to be stated once, thereby shortening names
  - Can also be used to generate a hybrid namespace
  - `typedef` can be used: `typedef Some::Thing thing;`
- A namespace can be defined more than once
  - Allows, for example, internal and external library definitions
Example

```
namespace Module1 { int x; }

namespace Module2 {
  inline int sqr(const int& i) {return i*i;}
  inline int halve(const int& i) {return i/2;}
}

using namespace Module1; //"import" everything

int main() {
  using Module2::halve; //"import" the halve function
  x = halve(x);
  sqr(x);               //Wrong
}
```
Linking C and C++ code

- The directive `extern "C"` specifies that the following declaration or definition should be linked as C, not C++ code:
  ```c
  extern "C" int f();
  ```
- Multiple declarations and definitions can be grouped in curly brackets:
  ```c
  extern "C" {
  int globalvar; //definition
  int f();
  void g(int);
  }
  ```
- Why do we need this? E.g. ‘Name mangling’ for overloaded functions.
User-defined types

- 
  C++ provides a means of defining classes and instantiating objects
- 
  Classes contain both data storage and functions which operate on storage
- 
  Classes have access control: private, protected and public
- 
  Classes are created with class or struct keywords
  - struct members default to public access; class to private
- 
  The constructor syntax is a member function with the same name as the class
- 
  The destructor syntax is a member function with the same name as the class, prefixed with a tilde (~)
- 
  A constructor can be overloaded to provide multiple instantiation methods
- 
  Can create static (i.e. per class) member variables
Example

```cpp
class Complex {
  double re, im;
  public:
    Complex(double r=0.0L, double i=0.0L);
};

Complex::Complex(double r, double i) {
  re = r, im = i;  // deprecated initialisation-by-assignment
}

int main() {
  Complex c(2.0), d(), e(1, 5.0L);
  return 0;
}
```
Constructors and destructors

- A default constructor is a function with no arguments (or only default arguments)
- If no constructor is specified, the compiler will generate one
- The programmer can specify one or more constructors
- Only one constructor is called when an object is created
- There can only be one destructor
  - This is called when a stack-allocated object goes out of scope or when a heap-allocated object is deallocated with `delete`; this also occurs for stack-allocated objects deallocated during exception handling (more later).
  - Stack-allocated objects with destructors are a useful way to release resources on scope exit (similar effect as Java try-finally) – “RAII: Resource Allocation is Initialisation”.

Copy constructor

- A new class instance can defined by assignment; for example;
  Complex c(1,2);
  Complex d = c;

- In this case, the new class instance is initialised with copies of all the existing class’ non-static member variables; no constructor is called

- This behaviour may not always be desirable (e.g. consider a class with a pointer as a member variable)
  - In which case, define an alternative copy constructor:
    Complex::Complex(const Complex&) { ... }

- If a copy constructor is not wanted, make the copy constructor a private member function, or in C++11 use delete.
Assignment operator

- By default an object is copied on assignment by overwriting all non-static member variables; for example:

```cpp
1 Complex c(), d(1.0, 2.3);
2 c = d; //assignment
```

- This behaviour may also not be desirable

- The assignment operator (`operator=`) can be defined explicitly:

```cpp
1 Complex& Complex::operator=(const Complex& c) {
2     ...  
3 }
```
Constant member functions

- Member functions can be declared `const`
- Prevents object members being modified by the function:

  ```
  1 double Complex::real() const {
  2    return re;
  3  }
  ```

- Logically gives `const Complex *this` instead of `Complex *this`
Arrays and the free store

- An array of class objects can be defined if a class has a default constructor.
- C++ has a `new` operator to place items on the heap:
  ```cpp
  Complex* c = new Complex(3.4);
  ```
- Items on the heap exist until they are explicitly deleted:
  ```cpp
  delete c;
  ```
- Since C++ (like C) doesn’t distinguish between a pointer to a single object and a pointer to an the first element of an array of objects, array deletion needs different syntax:
  ```cpp
  1 Complex* c = new Complex[5];
  2 ...
  3 delete[] c; // Cannot use "delete" here, only "delete[]"
  ```
- When an object is deleted, the object destructor is invoked. When an array is deleted, the object destructor is invoked on each element.
- The C++ standard library provides `std::vector` or `std::array`, so raw arrays are rarely used.
Exercises

1. Write an implementation of a class `LinkList` which stores zero or more positive integers internally as a linked list on the heap. The class should provide appropriate constructors and destructors and a method `pop()` to remove items from the head of the list. The method `pop()` should return -1 if there are no remaining items. Your implementation should override the copy constructor and assignment operator to copy the linked-list structure between class instances. You might like to test your implementation with the following:

```c
int main() {
    int test[] = {1,2,3,4,5};
    LinkList l1(test+1,4), l2(test,5);
    LinkList l3=l2, l4;
    l4=l1;
    printf("%d %d %d\n", l1.pop(), l3.pop(), l4.pop());
    return 0;
}
```

Hint: heap allocation & deallocation should occur exactly once!